UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/658,798	09/10/2003	Karsten K. Bohlmann	13913-153001 / 2003P00562	1815

32864 7590 05/09/2007

FISH & RICHARDSON, P.C.
PO BOX 1022
MINNEAPOLIS, MN 55440-1022

EXAMINER	
INGBERG, TODD D	

ART UNIT	PAPER NUMBER
2193	

MAIL DATE	DELIVERY MODE
05/09/2007	PAPER

Please find below and/or attached an Office communication concerning this application or proceeding.

The time period for reply, if any, is set in the attached communication.

Office Action Summary	Application No. 10/658,798	Applicant(s) BOHLMANN ET AL.	
	Examiner Todd Ingberg	Art Unit 2193	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☒ Responsive to communication(s) filed on 4/23/07.

2a)☒ This action is **FINAL**. 2b)☐ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) 1-50 is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☒ Claim(s) 1-19 is/are allowed.

6)☒ Claim(s) 50 is/are rejected.

7)☐ Claim(s) _____ is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☐ The specification is objected to by the Examiner.

10)☒ The drawing(s) filed on 10/25/2004 is/are: a)☒ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☒ Notice of References Cited (PTO-892)

2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)

3) ☐ Information Disclosure Statement(s) (PTO/SB/08) Paper No(s)/Mail Date _____.

4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____ .

5) ☐ Notice of Informal Patent Application

6) ☐ Other: _____.

DETAILED ACTION

Claim Status

Claims 1 – 50 have been examined.

Claims 1, 7, 12, 15, 17, 18, 24, 29, 32-35, 40, 42, 43, 45, 47 and 48 have been amended.

Claim 50 has been added.

Claim Rejections - 35 USC § 103

1. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

2. Claim 50 is rejected under 35 U.S.C. 103(a) as being unpatentable over SNAP by **Template** Software, Version 8.0 release in 1997, documentation set copyright date of 1998 in view of **ABAP** as taught by USPN #6,192,370 B1 Issued February 20, 2001 Primsch and XSLT Working with XML and HTML, Khun Yee Fung (referred to as **XML**) published December 28, 2000

Template teaches the use of a utility know as Pattern (SNAP, Chapter 7) where a programmer identifies input and specifies the output. The utility is intended to allow for transformations. Template teaches the programming language, SNAP. Template does not teach the programming language ABAP. It is SAP who teaches ABAP (ABAP, col 11, lines 62-67) and XML who teaches XML (XML). Therefore, it would have been obvious to one of ordinary skill in the art at the time of invention to take the Template SNAP utility, Pattern and define both input source definitions for both ABAP and XML and output targets for both XML and ABAP. Thus

allowing conversion between the two programming languages in both directions. Because, transforming programming languages can enable code to be reused with other tools.

50. (New) A method for transforming application data structures into an XML document, the method comprising: writing an application program having non-XML data structures; writing a transformation program; executing the application program; executing the transformation program when called for by the application program to transform the non-XML data structures from the application program into an XML document; and sending the XML document to a recipient.

Allowable Subject Matter

3. Claims 1 – 49 are allowed.

The scope of the claims for claims 1 – 49 is the ability to specifically convert from ABAP to XML and from XML to ABAP. ABAP is a proprietary language of the Assignee's.

Although, one of ordinary skill in the art could use the Template product SNAP, specifically the utility Pattern to convert from a form to another. The utility requires the conversion scheme to be set up. The claimed invention the utility has the encoded scheme integrated as a specific utility. The Examiner can not reasonably state it would have been obvious to one of ordinary skill in the art to integrate the conversion scheme. Examiner determines the SNAP products utility Pattern is flexible and would be the obvious choice so the utility could be reused.

Examiner's Remarks

4. The amendment to the Specification indicating the Domestic priority date has been entered.

Correspondence Information

5. Any inquiry concerning this communication or earlier communications from the examiner should be directed to Todd Ingberg whose telephone number is (571) 272-3723. The examiner can normally be reached on during the work week..

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Meng-Ai An can be reached on (571) 272-3756. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free). If you would like assistance from a USPTO Customer Service Representative or access to the automated information system, call 800-786-9199 (IN USA OR CANADA) or 571-272-1000.

Todd Ingberg
Primary Examiner
Art Unit 2193

TI

SNAP

 Foundation Template

Using the SNAP Language

TEMPLATE
SOFTWARE

Using the SNAP Language

SNAP Version 8.0

Template Software, Inc.
45365 Vintage Park Plaza, Suite 100
Dulles, VA 20166

License Information

This document describes Release 8.0 of SNAP. This document is subject to change without notice, and Template Software assumes no responsibility for any errors that may appear in this document. The SNAP software described in this document is furnished under a license agreement. It is unlawful to copy SNAP on any medium for any purpose other than the licensee's use.

Trademark Acknowledgments

SNAP is a registered trademark of Template Software, Inc. The Workflow Template is a trademark of Template Software, Inc.

Borland C++ is a trademark of Borland International, Inc. Alpha AXP, AXP, DEC, DECnet, DECstation, DECwindows, Open VMS AXP, VAX, VAXstation, VMS, and ULTRIX are trademarks of Digital Equipment Corporation. HP and HP-UX are registered trademarks of Hewlett-Packard Company. INFORMIX is a registered trademark of Informix Software, Inc. DATABASE2, DB2, IBM, and OS/2 are registered trademarks of International Business Machine Corporation. AIX, C Set++, and RISC System/6000 are trademarks of International Business Machine Corporation. IXI Motif is a trademark of IXI Limited. X Window System is a trademark of Massachusetts Institute of Technology. Microsoft, MS, and MS-DOS are registered trademarks of Microsoft Corporation. Visual C++, Windows, Windows NT, WIN32, and WIN32s are trademarks of Microsoft Corporation. Motif and OSF/1 are registered trademarks of Open Software Foundation, Inc. ORACLE, Oracle7, and SQL*Plus are registered trademarks of Oracle Corporation. Pro*C is a trademark of Oracle Corporation. OSx and Pyramid are registered trademarks of Pyramid Technology Corporation. DC is a trademark of Pyramid Technology Corporation. Solaris, Sun, SunOS, and SPARC are trademarks of Sun Microsystems, Inc. SYBASE is a registered trademark of Sybase, Inc. DB-Library, Open Client, SQL Server, and SQL Toolset are trademarks of Sybase, Inc. MultiNet is a registered trademark of TGV, Inc. SCO is a trademark of The Santa Cruz Operation. Unisys is a trademark of Unisys Corporation. UNIX are registered trademarks of X/Open Company Limited.

Copyright Notice

If you have any comments concerning this document or software, please forward them to;

Template Software, Inc.
45365 Vintage Park Plaza, Suite 100
Dulles, VA 20166
Internet address: support@template.com

CHAPTER 7

USING PATTERNS

About this chapter

This chapter describes how to manipulate text by matching parts of the text to a pattern.

Contents

Introduction

You might want to design an application that can examine the form of string expressions such as attribute values or class object names. You might need to search for occurrences of particular character strings and to extract portions of those strings. For example, you might want to search for attributes having certain values, and display those values to the end user. Sometimes you might want to display the attribute values minus their certainty factors and thus find it necessary to strip the certainty factor from the attribute value.

SNAP provides a pattern matching facility that enables you to perform these matching and extracting tasks. Using pattern matching, you can extract parts of any string matching a specified form. Pattern matching compares a pattern defined in a class with a string. A SNAP pattern is a template that can be used to determine if a string matches a particular form. If a pattern match is successful, you can easily extract portions of the string.

SNAP enables you to:

- Define a **pattern** in a class.
- Test whether a given string matches (fits the form of) the pattern.
- Extract parts of a string that matches a pattern.

These capabilities, collectively called **string pattern matching**, have many uses. For example, you might expect data for a patient record to be a string in the following form:

```
Name,Street Address,City,State,Zip Code
```

More specifically, you know the following about the form of the data in these records:

- Each part of the string is separated by either one comma or one asterisk.
- Commas and asterisks are used nowhere else in the string.
- *Name* is no more than 25 characters.
- *Street Address* is no more than 30 characters.
- *City* is no more than 20 characters.
- *State* is two characters.
- *Zip Code* is five numeric characters.

Knowing that many such records occur in this form, you can define a pattern that describes this form. If you are interested in a portion of the data for each record, you can then extract that portion from each string matching the form of a record pattern. You can also detect strings that do not have this form because they will not match the pattern.

The rest of this chapter discusses:

- What kinds of applications pattern matching is commonly used for.
- How to define a pattern.
- How to use the *match* function to match a string with a pattern.
- How to use the *extract* function to extract part of a matching string.

How pattern matching is performed

After defining your patterns, the actual pattern matching is done by using two functions—*match* and *extract*—that enable you to use pattern matching in your application. The *match* function tests whether a string matches a pattern. The *extract* function extracts a portion of a matching string and can be used only after the *match* function has determined whether the string matches the pattern.

Uses for pattern matching

Use pattern matching whenever your application needs to interpret the form of strings. The following sections discuss two uses of pattern matching.

Using pattern matching to interpret input values

When attribute values are strings that contain sets of items in a known form, you can use pattern matching to extract parts of the strings. For example, you can define a pattern to represent the form of a patient record like the one in the introduction. You can use an *ask-for* function to query the end user for the value of a *str* attribute in which the record's knowledge will be stored. The end user would enter a string containing knowledge for the patient's name, street address, city, state, and zip code. You can then match the value of the attribute with the record pattern. If the value of the *str* attribute holding the data matches this pattern, the string's form is verified and its parts (the data for each field of the patient record) can be easily extracted and used.

Using pattern matching to format output values

You can also use pattern matching to format parts of strings to display to the end user. For example, suppose that you want to display each part of the patient record in the previous example on a different line. You can match the string with the patient record pattern as described preceding, then extract each part of the string and assign it to a *str* attribute to store the value for later use. When you want to display the parts of the string, use the *message* command to display the value of each of these attributes on a different line. You can also extract and then display the parts of the string directly through the *message* command if you do not need to save their values for other uses.

Defining patterns

You define a pattern in a class of your SNAP application. To define a pattern, you must:

- Decide what set of strings you want the pattern to be able to match.
- Define a pattern so that only this set of strings will match it.

The format of a pattern definition is as follows:

```
public or protected
pattern pattern name
{
   pattern elements
}
{attachment}...{attachment};
```

Argument	Description
public or protected	*public* – Indicates that the pattern can be accessed from anywhere in the application. *protected* – Indicates that the pattern can be accessed only from within its class and subclasses. If neither *public* nor *protected* is specified, the pattern is private—it can be accessed only from within its class.
pattern name	A name that you assign to identify the pattern in the application.
pattern elements	The elements that you want strings to match. Each element corresponds to a portion of the string to be matched. A pattern element can contain characters, pattern symbols, names of other patterns, and pattern components. See *Format of pattern elements* on page 7-6 for more information.
attachment	A text attachment. See *Attachments* on page 4-33 in *Chapter 4, Fundamentals of the SNAP Language*, for more information.

When SNAP tries to match a string with a pattern, it checks each part of the string to see if it matches the elements defined in the pattern. A string matches a pattern if the string can be broken down into parts such that each part of the string exactly matches the defined sequence of pattern elements. The next section describes how to define a pattern element.

Defining pattern elements

A pattern element can be simple or complex. The pattern element in the following pattern is simple:

```
pattern patternl {"abc"};
```

The pattern element in this pattern is the literal string *"abc"*. The only string that can match this pattern is:

```
abc
```

The next pattern element in the following pattern is more complex:

```
pattern pattern1 {[abc]};
```

This pattern element is a set of alternative characters, *[abc]*, which means that a string consisting of any one character in the set can match the pattern. The only strings that can match this pattern are:

a or b or c

In the following examples, the pattern elements are again sets of alternative characters. However, instead of enumerating each possible character in the set, a range of alternative characters is expressed by using a hyphen:

```
pattern Uppercase {[A-Z]};
pattern Alpha {[A-Za-z]};
```

The hyphen between the *A* and the Z in the *Uppercase* pattern indicates that any single uppercase letter from *A* to Z (including *A* and Z) will match this pattern element. In the *Alpha* pattern, the range of possible characters is any upper-case letter from *A* to Z or any lowercase letter from *a* to *z*.

Finally, the following pattern definition is even more complex. It uses a conjunction of pattern elements—the elements are the names of two other patterns, *Uppercase* and *Alpha*. It also uses a repetition range, *<1-5>*. A repetition range specifies the number of times the preceding pattern element can be repeated. In this example, the repetition range specifies that the element *Alpha* can be repeated from one to five times:

```
pattern Name {Uppercase & Alpha<1-5>};
```

To match the pattern *Name*, a string must match both elements: *Uppercase* and *Alpha*. The string must begin with an uppercase letter and can be followed by one to five uppercase or lowercase letters. Thus, the string *Heidi* would match the pattern *Name* while the string *heidi* would not.

You can define patterns that are much more complex than the ones shown here. For example, you can explicitly specify that certain characters not be matched. You can divide a pattern element into components and name the components so that you can extract them later. A pattern element can itself contain pattern elements. The formats and symbols used to define pattern elements are described in the next subsection.

Format of pattern elements

Table 7-1 shows the formats you can use to define pattern elements:

Table 7-1. Formats of pattern elements

Format	Description
string	A literal string.
string constant	A named string constant defined in the class.
# (wildcard)	A wildcard that represents any single character when used in a pattern definition.
[set of alternative characters]	A set of characters; cannot span more than one line.
pattern component name = pattern element	A named pattern element.
pattern name	The name of another pattern defined in the class. A pattern referenced in another pattern cannot contain components.
pattern element<repetition range>	A repetition of pattern elements of any format. The *repetition range* indicates the number of times that the preceding element can be repeated.
pattern element & pattern element	A concatenation of pattern elements of any format.
pattern element | pattern element	A list of alternative pattern elements of any format.
(pattern element)	A pattern element of any format.

Table 7-2 shows examples of using pattern elements:

Table 7-2. Examples of pattern elements

Pattern element	Example pattern definition
string	Pattern Name: "hello"
string constant name	Pattern Name: First where *First* is the name of a string constant
# wildcard	Pattern Name: #
[set of alternative characters]	Pattern Name: [abc]
	Pattern Name: [A-Za-z]
	Pattern Name: [^A-Za-z]
	Pattern Name: [-!#$%^&*()={}[\]]
pattern component name = pattern element	Pattern Name: Initials = [A-Z]<2-3> where *Initials* in the name of a pattern component
pattern name	Pattern Name: Alpha where *Alpha* is the name of another pattern
pattern element<repetition range>	Pattern Name: #<1-5>
pattern element & pattern element	Pattern Name: [abc] & "1"
pattern element | pattern element	Pattern Name: "abc" | [0-9]
(pattern element)	Pattern Name: (First Name & Last Name)

Note that some formats allow you to nest a pattern element within a pattern element. In addition, you can use symbols such as #, *, &, |, and < > in pattern elements. All of the formats and symbols are described in more detail in the next section.

Matching specific strings

If you want to match a specific string of characters, define a pattern element using a string or a string constant.

Defining strings

To match a specific string of characters, define a pattern element that is a string. The matching string must match every character in the string character for character, except for special characters. For example, the following pattern is defined using a string:

```
pattern pattern1 {"hello"};
```

The only string that matches this pattern is:

```
hello
```

Defining string constants

Another way to match a specific string of characters is to define a pattern element that is a string constant. As with a string, the matching string must match every character in the string represented by the string constant. For example, the following pattern is defined using a string constant:

```
pattern pattern1 {greeting};
```

If the string constant *greeting* is defined to be *"hello"*, the only string that matches this pattern is:

```
hello
```

Matching single characters

If you want to match a single character, you can define a pattern element using one of three formats. You can define an element that is composed of:

- The wildcard character (#).
- A set of alternative characters that specifies a group of character that can be matched.
- A set of alternative characters that specifies a group of character that cannot be matched.

Each format is described in the following sections.

Defining wildcards in patterns

To match any single character, define a pattern element that uses a wildcard. In pattern elements, the wildcard character is the pound (#) character. For example, the following pattern is defined using a wildcard:

```
pattern pattern1 {#};
```

Any string containing a single character will match the preceding pattern, for example, *a* or the Escape character.

In the following example, because of the repetition range, a string of any length containing any characters will match the pattern:

```
pattern pattern2 {#<0-*>};
```

Repetition ranges are described later in this chapter.

You can use the wildcard in other ways. For example, if you want to check whether a string contains the characters *abc* but you don't know where this may occur in the string, you can define the following pattern:

```
pattern pattern3 {#<0-*> & "abc" & #<0-*>};
```

You can also use the wildcard to pick up alternate spellings or variable endings.

Defining sets of alternative characters

To match one character out of set of characters, define a pattern element that is a set of alternative characters. A set of alternative characters includes all the characters within its brackets. The matching string can consist of any single character in the set. For example, the following pattern is defined using a set of alternative characters:

```
pattern pattern4 {[abc]};
```

The strings that match this pattern are:

`a` or `b` or `c`

If you want to define a range of possible characters rather than enumerating every possibility, you can do so by specifying the lower and upper bounds of a range and separating them by a hyphen. You can define ranges of letters and digits. For example, the following pattern is defined with a range of letters and digits:

```
pattern pattern5 {[a-c1-2]};
```

The strings that match this pattern are:

`a` or `b` or `c` or `1` or `2`

Uppercase letters will not match this pattern. Note that you cannot specify a range of numbers. For example, the following pattern does not define a range of numbers from 1 to 50:

```
pattern pattern6 {[1-50]};
```

The strings that will match this pattern are *1*, *2*, *3*, *4*, *5*, and *0*. But the numbers *6*, *7*, *8*, *9*, etc., will not match.

You do not need to enclose characters in a set of alternative characters in quotation marks. When interpreting a set of alternative characters, SNAP interprets characters as literal characters with two exceptions:

- The first exception is a hyphen between two characters.

  This is not interpreted as a literal character; instead, SNAP interprets it as indicating a range of characters. The character on the left side of the hyphen specifies the lower bound of the range and the character on the right side specifies the upper bound of the range. The range includes all characters in the character set between and including the specified characters. For example, the character set *[A-Z]* does not represent the three literal characters *A*, -, and *Z* but represents all the uppercase letters in the alphabet. Lowercase letters will not match this range.

  If you want a hyphen to be interpreted as a literal character within a set of alternative characters, it must appear at the beginning of the character set as follows:

  ```
  pattern pattern7 {[-AZ]};
  ```

  The strings that match this pattern are:

  ```
  - or A or Z
  ```

  When defining a range of alternative characters, the characters on both sides of the hyphen must be of the same type—either alphabetic or numeric. Furthermore, when defining a range of alphabetic characters, the characters must either be both lowercase or both uppercase. The letter specified for the lower bound must alphabetically precede the upper bound letter. When defining a range of numeric characters, the digit on the left side must be less than the number on the digit side.

  The following patterns are examples of illegal definitions:

  ```
  pattern patterna {[A-z]};
  pattern patternb {[1-*]};
  pattern patternc {[a-*]};
  pattern patternd {[1-a]};
  pattern patterne {[b-a]};
  pattern patternf {[a-]};
  ```

- The other exception is when specifying backslashes and right brackets as literal characters.

  To specify a backslash or right bracket as a literal character in a set of alternative characters, you must precede them with a backslash:

  ```
  pattern pattern8 {["\\.#\],<]};
  ```

  The strings that match this pattern are:

  ```
  " or \ or . or # or ] or , or <
  ```

  Notice that you do not need to precede a quotation mark with a backslash; SNAP interprets a quotation mark in a set of alternative characters as a literal character. The # is also interpreted as a literal character.

Chapter 7

Defining characters as not in sets of alternative characters

To match all characters but a few specific characters, define a set of alternative characters that contains only the characters you do not want matched. When a caret precedes the characters in the set, it indicates that any but the following characters can be matched. All other characters are implicitly allowed. For example, the following pattern is defined using characters not in the set of alternative characters:

```
pattern pattern9 {[^a-zA-Z]};
```

Any string containing a single non-alphabetic character will match the preceding pattern, for example 3. In the next pattern, any character besides a hyphen will match:

```
pattern pattern10 {[^-]};
```

Any string containing any single character that is not a hyphen will match the preceding pattern, for example, *A* In the next pattern, any non-numeric character will match:

```
pattern pattern11 {[^0-9]};
```

If the caret appears anywhere in the character set, SNAP interprets it as a literal character. It must appear at the beginning of the set to indicate *not in set*. For example, SNAP interprets the caret in the following pattern as a literal character:

```
pattern pattern12 {[-a^]};
```

The strings that match this pattern are:

`-` or `^` or `a`

Matching combinations of elements

You can define more complex patterns by combining pattern elements of same or different formats. You can:

- Define patterns that are a disjunction of pattern elements.
- Define patterns that are a conjunction of pattern elements.
- Repeat elements in a pattern

The following sections discuss formats that allow you to define complex patterns.

Defining disjunctions of elements

Another way to indicate alternative sets of characters is to define a pattern element that is a disjunction of elements. A disjunction of elements is an element followed by a bar sign (|) followed by another element where one of the elements must be matched. For example, the following pattern is defined using a disjunction of three elements:

```
pattern pattern13 {"aa" | "cd" | "ef"};
```

The strings that match this pattern are:

`aa` or `cd` or `ef`

The elements in a disjunction do not have to be of the same type. For example, you can define the following pattern:

```
pattern pattern14 {"ab" | [cd]};
```

The strings that match this pattern are:

`ab` or `c` or `d`

The pattern elements in the following pattern definitions are equivalent, that is, the same character strings will match both pattern elements:

```
pattern pattern15 {[abc]};
pattern pattern16 {"a" | "b" | "c"};
```

The strings that match both of these patterns are:

`a` or `b` or `c`

Defining conjunctions of elements

To form a more complex pattern by joining pattern elements, define a pattern element that is a conjunction of elements. A conjunction of elements is an element followed by an ampersand (&) followed by another element where both elements must be matched. The following pattern is defined using a conjunction of elements:

```
pattern pattern17 {"aa" & "cd"};
```

The string that matches this pattern is:

```
aacd
```

As with a disjunction, the elements in a conjunction do not have to be of the same form. For example, you can define the following pattern:

```
pattern pattern18 {"c" & [123]};
```

The strings that match this pattern are:

`c1` or `c2` or `c3`

Defining repetitions of elements

To define a pattern that matches a given element that might appear repeatedly, modify the element with a repetition range. A repetition range is a numeric range enclosed in angle brackets. A repetition range allows you to express that an element be repeated a certain number of times. For example, the element in the following pattern is modified by a repetition range that specifies that the element can be matched zero to two times:

```
pattern pattern19 {"hi"<0-2>};
```

The strings that match this pattern are:

`""` or `hi` or `hihi`

Note that allowing an element to be repeated zero times means that it will match an empty string.

Format of repetition ranges

A repetition range must always be expressed in terms of a numeric range. The format of a repetition range is as follows:

```
<lower end of range  –  upper end of range>
```

Argument	Description
lower end of range	A non-negative number that specifies the least number of times that the element can be matched to constitute a match.
upper end of range	A number that specifies the most number of times that the element can be matched to constitute a match. This number must be greater than or equal to the number specified for *lower end of range*. An asterisk can also be specified to indicate no upper bound.

An element modified by a repetition cannot contain a component. Components are explained in more detail later in the section.

In the following example, a repetition range is used to specify that the matching string must contain two repetitions of the literal string *Supercalifragilisticexpialidocious*:

```
pattern pattern20 {"Supercalifragilisticexpialidocious"<2-2>};
```

The only string that matches this pattern is:

```
SupercalifragilisticexpialidociousSupercalifragilisticexpialidocious
```

In the following example, a repetition range is used to specify that the matching string must be exactly three characters long; the characters may be any combination of `a` and `b`:

```
pattern pattern21 {[a-b]<3-3>};
```

The strings that match this pattern are:

`aaa` or `aab` or `aba` or `abb` or `baa` or `bab` or `bba` or `bbb`

The preceding pattern could not be specified as the following:

```
pattern pattern22 {[ab]<3>};
```

A repetition range must always contain a lower and upper bound.

Note that a repetition range applies only to the preceding element. For example, the repetition range in the following order applies only to the second literal character:

```
pattern pattern23 {"a" & "b"<1-2>};
```

The strings that match this pattern are:

`ab` or `abb`

The following strings do not match this pattern:

```
abab or aab or aabb
```

Defining an unspecified number of repetitions

To allow an unspecified maximum number of repetitions, specify an asterisk as the upper range. For example, there is no limit on the number of times that the element *ab* in the following pattern can be repeated, but it must repeat at least twice:

```
pattern pattern24: "ab"<2-*>;
```

The strings that match this pattern are:

`abab` or `ababab` or `abababab` or ...

Chapter 7

Matching more complex patterns

You can define even more complex patterns by:

- Defining patterns in which some elements are assigned to named components.
- Defining patterns that reference other patterns.
- Using parentheses to group pattern elements.

The following sections discuss formats that allow you to even more complex patterns.

Defining pattern components

By naming an element, you can later extract the string that matches that element. A named element is called a **pattern component.** A pattern component consists of a component name followed by an equals sign followed by a pattern element. Naming an element does not change what strings can match that element—a pattern component is no different from any other element except that it has a name and thus can later be used with the *extract* function to extract the part of the string that matched the component.

You do not have to name all of the elements in a pattern definition, just those elements whose matching strings you want to extract. (In fact, for optimal performance, we recommend that you only name components that you need to extract.) You can then access the string that matches the component by using the *extract* function.

The following pattern is defined using a pattern component called *Comp1*:

```
pattern25 {Compl = "a"<0-*> & "b"};
```

The strings that match this pattern are:

```
b or ab or aab or aaab or ...
```

Notice that the *& "b"* is not part of the component definition and thus, the portions of the preceding strings that match *Compl* are:

```
a or aa or aaa or ...
```

If you want the component definition to include *& "b"*, add parentheses to group the elements as shown in the following pattern:

```
pattern pattern26 {Compl = ("a"<0-*> & "b")};
```

The strings that match this pattern are:

```
b or ab or aab or aaab or ...
```

The strings that match *Compl* are the same as the preceding strings. The parentheses change the precedence in which SNAP evaluates the pattern. Precedence and using parentheses in pattern definitions are discussed later in the chapter.

Note that if a component is defined using another component, the second component must be enclosed in parentheses. For example, in the following pattern,

```
pattern Patient Code
{Patient ID = (Hospital Code = [A-Z]<3-3> & #<5-5>)};
```

The component *Patient ID* is composed of two elements: a component called *Hospital Code* and the element #<5-5>. If desired, you could extract the component *Hospital Code* (which is a three-letter uppercase code) from the component *Patient ID* (which is a 8-letter code).

Also note that an element modified by a repetition range cannot contain a component. For example, the following *pattern27* is illegal while the following *pattern28* is legal:

```
pattern pattern27 {(compl = "ab")<1-2>};   // illegal
pattern pattern28 {comp2 = ("ab"<1-2>)};   // legal
```

In the first example, the entire element which is a component is modified by a repetition while in the second example, only the component definition is modified by a repetition. Entire components cannot be modified by a repetition range because if there are multiple instances of the component, it will be unclear which component is referenced.

Defining pattern elements using another pattern

You can define more complex patterns by referencing another pattern within a pattern definition. You can define generic patterns that can be used in other patterns. For example, the following pattern *Name* references two other patterns: *Uppercase* and *Alpha*:

```
pattern
Uppercase {
   [A-Z]
}

pattern
Alpha {
   [A-Za-z]
}

pattern
Name {
   Uppercase &
   Alpha <0-*>
}
```

An example of a string that could match this pattern is:

```
Doug
```

In a more complex version of this example, the pattern *Full Name* is composed of three pattern components: *First Name, Init,* and *Last Name.* Each pattern component references another pattern.

```
pattern
White Space {
   [ \t\n] <1-*>
}

pattern
Uppercase {
   [A-Z]
}

pattern
Alpha {
   [A-Za-z]
}

pattern
Initial {
   Uppercase &
   "."
}

pattern
Name {
   Uppercase &
   Alpha <0-*>
}

pattern
Full Name {
   First Name = Name &
   White Space &
   Init = Initial &
   White Space &
   Last Name = Name
}
```

The pattern components *First Name* and *Last Name* both reference the pattern *Name* and the pattern component *Init* references the pattern *Initial.* An example of a string that matches the pattern *Full Name* is:

```
Doug J. Witten
```

If you then extract the value for each of the three components, the value of *First Name* is *Doug,* the value of *Init* is *J.,* and the value of *Last Name* is *Witten.* As you can see, referencing another pattern within a pattern definition allows you to define generic patterns that can be used to define more complex patterns.

Note that a referenced pattern cannot contain components.

SNAP pattern symbols

Table 7-3 summarizes the symbols that you can use in SNAP patterns:

Table 7-3. Pattern symbols

Symbol	Meaning	Example pattern	Matching string
" "	Indicates a literal character or string.	p1: "aa"	`aa`
#	A wildcard that matches any character.	p2: #<0-*>;	`A string of any length and containing any characters.`
[]	Indicates a set of alternative characters.	p3: [abc];	`a or b or c`
[^]	When it is the first character in a set of alternative characters, indicates all but the following characters are in the set[a].	p4: [^a-zA-Z]	`Any single non-alphabetic character`
-	Indicates a range within a set of alternative characters or within a repetition[b].	p5: [a-b1-2]; p6: "hi"<0-2>;	`a or b or 1 or 2` `"" or hi or hihi`
< >	Indicates repetition of a pattern element.	p7: "hi"<2-2>;	`hihi`
*	Within a repetition, indicates any number of repetitions.	p8: "ab"<2-*>;	`abab or ababab or abababab...`
&	Indicates a conjunction of pattern elements.	p9: "aa" & "cd";	`aacd`
|	Indicates a disjunction of pattern elements.	p10: "aa" | "cd";	`aa or cd`
=	Indicates a pattern component. The name preceding = is the component name and the element following is the pattern component[c]	p11: comp = "a"<0-*> & "b";	`b or ab or aab or aaab...`
( )	Indicates explicit grouping for a pattern element.	p12: ("ab" & "cd")<2-2>;	`abcdabcd`

a. If the caret does not appear at the beginning of a character set, SNAP interprets it as a literal character. It must appear at the beginning to indicate *not in set*.

b. If the hyphen appears at the beginning of a set of alternative characters, SNAP interprets the hyphen as a literal character. It must appear between two characters to indicate a range. If the caret appears appears at the beginning of a set of alternative characters, it indicates that the following characters are characters not to be matched. If any of the other pattern symbols appear inside the set, they are interpreted as literal characters.

c. An element modified by a repetition cannot contain a component. For example, *pattern1* below is legal while *pattern2* is illegal:

```
pattern pattern1 {comp1 = ("ab"<1-2>)};   // legal
pattern pattern2 {(comp2 = "ab")<1-2>};   // illegal
```

Precedence of pattern symbols

The precedence of a symbol determines the order in which SNAP evaluates the characters and symbols in a pattern element. SNAP uses the following order of precedence to evaluate pattern definitions:

1. ()
2. <>
3. =
4. &
5. |

For example, in the following pattern, the elements on either side of the ampersand are "more tightly bound" because the ampersand has precedence over the bar. Thus, *"a"& "b"* in the following pattern are more tightly bound than *"b" | "c"*:

```
pattern pattern29 {"a" & "b" | "c"};
```

The two strings that match this pattern are:

```
ab or c
```

However, the string *ac* does not match this pattern.

Using parentheses to establish precedence

You can use parentheses in pattern elements to change the precedence. Parentheses indicate explicit grouping for a pattern element. In the following example, parentheses have been added to the previous example. Notice the difference in the matching strings:

```
pattern pattern30 {"a" & ("b" | "c")};
```

The two strings that match this pattern are:

```
ab or ac
```

However, the string *c* does not match this pattern.

The following pattern contains a component and a repetition range. The repetition range has the highest precedence in this pattern, followed by the equal sign, followed by the ampersand. Thus, the repetition range modifies only the *b* and the component is defined to be *Compl = "a"*.

```
pattern pattern31 {Compl = "a" & "b"<1-2>};
```

Although parentheses are not specified in the preceding pattern, they are implicitly specified by the rules of precedence. You may find it helpful to specify parentheses to clarify precedence, although technically it is not necessary unless you want to change the precedence.

```
pattern pattern32 {(Compl = "a") & ("b"<1-2>)};
```

The two strings that match the preceding patterns are:

```
ab and abb
```

The string that matches *Compl* is:

```
a
```

If you add parentheses to change the precedence, the repetition range now modifies the entire component rather than just the *b*:

```
pattern pattern33 {Compl = ("a" & "b")<1-2>};
```

The two strings that match this pattern are:

```
ab and abab
```

The portions of the strings that match *Compl* are:

```
ab and abab
```

Note that you can also add parentheses for clarity without affecting the precedence. For example, the following two patterns are equivalent:

```
pattern pattern34 {compl = "a" & "b"};
pattern pattern35 {(compl = "a") & "b"};
```

The parentheses in *pattern35* just make it clearer that that the definition of *Compl* consists only of the literal string *"a"* and does not include *& "b"*. However, the parentheses in the following pattern do change the precedence:

```
pattern pattern36 {compl = ("a" & "b")};
```

Compl now consists of the elements *"a" & "b"*.

Also note that, as described previously, an entire component cannot be modified by a repetition range. For example, the following pattern definition is not allowed:

```
pattern pattern37 {(Compl = "a" & "b")<1-2>};
```

The next section describes pattern symbols in more detail.

Defining patterns

When you know the format of strings you want to match, you can define a pattern to match them, as described in *Defining patterns* on page 7-4.

For example, consider how you would define a pattern to match strings containing patient records like those described in the example in the introduction. First, gather all available information about the form of patient records. From the example, you know that each record string has five parts: *Name, Street Address, City, State,* and *Zip Code.* Assume the following:

- Each part of the string is separated by either one comma or one asterisk.
- Commas and asterisks are used nowhere else in the string.
- *Name* is no more than 25 characters.
- *Street Address* is no more than 30 characters.
- *City* is no more than 20 characters.
- *State* is two characters.
- *Zip Code* is five numeric characters.

The following pattern definition matches patient records of the preceding form:

```
pattern
Patient Record {
   Patient Name = # <1-25> &
   [, *] &
   Street Address = # <1-30> &
   [,*] &
   City = # <1-20> &
   [, *] &
   State = # <2-2> &
   [, *] &
   Zip Code = [0-9] <5-5>
}
{purpose: " This pattern matches patient records"}
```

It is often useful to define a complex pattern in steps. First, define a fairly simple "prototype" pattern so that strings close to the "complex" form will match the pattern. Then, add or change specifications one at a time so that strings increasingly closer to the complex form match the pattern. This approach often minimizes the time you spend developing (and debugging) the pattern.

For example, you could begin defining the preceding pattern by defining an element that checks for the patient's name, testing that element, and if it works, define the next element (the delimiters), testing that, and so forth. After defining all of the element, you can decide which elements you want to be able to extract and name those elements.

You could change the definition of some elements to further qualify what characters must be contained in each part of the patient record. For example, you could restrict the characters that match the *Name, Street Address, City, State* to be alphabetic characters. A further step could be to modify the component *State* by replacing the #<2-2> with a disjunction of literal strings (e.g. *"AL"* | *"AK"* | *"AZ"...*) that contain the 50 two-letter state abbreviations.

Notice that the preceding pattern uses two methods of limiting strings: repetition ranges to define a fixed field length, and elements that check for delimiters in the form of commas and asterisks.

Matching and extracting

Now that you have learned how to define a pattern, you can use the predefined SNAP functions *match* and *extract* to perform the pattern matching.

The *match* function

You use the *match* function to compare a string with a pattern defined in the class. The format of the *match* function is as follows:

```
match (pattern name, string expression)
```

Argument	Description
pattern name	The name of the pattern that string should match.
string expression	An expression that evaluates to a string.

The *match* function returns *true* if the string matches the pattern and *false* if it does not. Use this function in conditions (with rules, demons, *if* statements, *while* statements) where its return value can be tested. For example, suppose that a *str* attribute called *Current Record* has the following value:

```
Doug J. Witten,5902 Shirlington Lane, Manassas,VA,22110
```

Using the *Patient Record* pattern discussed in the previous section, the following *match* function returns *true*:

```
match (Patient Record, Current Record)
```

The value of the attribute *Current Record* is a string that matches the pattern defined by *Patient Record*.

Chapter 7

The *extract* function

You use the predefined SNAP function *extract* to extract a portion of a matching string. The *extract* function returns the value of a pattern component of a string that has been successfully matched. The returned value is a string. The format of the *extract* function is as follows:

```
extract (pattern name, pattern component name)
```

Argument	Description
pattern name	The name of the pattern containing the component to be extracted.
pattern component name	The name of a pattern component whose value is to be extracted.

Using the *Patient Record* pattern discussed in the previous section, the following *extract* function returns *Doug J. Witten* for the value of the component *Patient Name*:

```
extract (Patient Record, Patient Name)
```

Before you use *extract* with a pattern, make sure that a string has successfully matched the pattern. To do this, test the return value of the *match* function. The following example uses an *if* statement to test the result of *match* before using *extract* to retrieve components' values from the *Patient Record* pattern:

```
if( match( Patient Record, Current Record)) {
   message "The patient's name is " + extract( Patient Record,
                                              Patient Name);
}
else {
   message "The match attempt was unsuccessful." +
   "Please check the patient record's form.";
}
```

This *if* statement verifies that the *Patient Record* pattern was matched before the value of one of its components is extracted. If *match* returns *true*, then the application displays the name of the patient; otherwise, it displays an error message to inform the end user that the pattern matching attempt was unsuccessful.

If a pattern used with *extract* has been matched more than once in the same session, the *extract* function returns component values from the most recent of these matches. If the pattern has not been previously matched, the *extract* function returns an empty string.

The following example performs patterning matching using both the *match* and *extract* functions. The example tries to match an employee name stored in the attribute *Employee Name* with the pattern *Full Name*. If the employee's name matches the pattern, each part of the name is extracted from the string and displayed in the following order: *Last Name, First Name, Init.*

```
pattern
White Space {
   [ \t\n] <1-*>
}

pattern
Uppercase {
   [A-Z]
}

pattern
Alpha {
   [A-Za-z]
}

pattern
Initial {
   Uppercase &
   "."
}

pattern
Name {
   Uppercase &
   Alpha <0-*>
}

pattern
Full Name {
   First Name = Name &
   White Space &
   Init = Initial &
   White Space &
   Last Name = Name
}

str                              Employee Name
   [default: askfor];

void
Function1()
{
   this.Employee Name = "Fred R. Jones";

   if( match( Full Name, this.Employee Name)) {
      message extract( Full Name, Last Name) + ", " + extract( Full Name,
                                                                First Name)
            + " " + extract( Full Name, Init);
   }
}
```

This application displays the following message:

```
Jones, Fred R.
```

TEMPLATE
SOFTWARE

45365 Vintage Park Plaza, Suite 100 Dulles, Virginia 20166 703-318-1000